Poster: Design Patterns for Multithreaded Software Pipelines in Real Time Applications

Stefan Preuss
University of Karlsruhe, IBDS
Am Fasanengarten 5
76128, Karlsruhe, Germany
stpreuss@ira.ika.de

Alfred A. Schmitt
University of Karlsruhe, IBDS
Am Fasanengarten 5
76128, Karlsruhe, Germany
aschmitt@ira.ika.de

ABSTRACT

This paper presents design patterns that will help in the task of parallelizing graphical real time algorithms, according to the example of a visual real time 3D reconstruction algorithm. These algorithms can often be designed as a dataflow graph, so they can be coarsely granular parallelized in a pipeline pattern. With these patterns, the design process of the parallelization is detached from the design of the graphical algorithm. The advantages and drawbacks of these patterns are discussed with regard to speed, but also to handling and error-proneness and the demanded robustness of real time applications, due to the varying workload of the different steps or data loss or obsoleteness during processing.

Keywords

parallelization, real time application, software architecture.

1. INTRODUCTION

Decent workstations are often based on SMP systems and recently even multicore processors are becoming affordable. One great advantage of these UMA designs is the low cost of communicating complex data structures (i.e. 3D meshes) between threads. Many graphical algorithms can be designed as processing pipelines similar to dataflow oriented processors and thus be parallelized coarsely granular, in order to get a faster response and/or higher output frequency of results. The pipeline approach results mostly (but not exclusivly) in the latter.

The danger of such an attempt arises from the indeterministic process flow. Deadlocks, random data order, changing or deletion of data in use by other elements must be considered. Finding the resulting errors is hard and time consuming. The use of the presented design patterns will help to minimize these dangers, that occur from parallelizing an algorithm by giving each thread a clearly defined area of responsibility.

2. MOTIVATION

The pipeline parallelization patterns resulted from the parallelization of our reconstruction algorithm. It is based on the works of [Lau94] and [Fau03]. [Lau94] introduced the concept of a visual hull and, based on this, [Fau03] developed his automatic reconstruction process of real objects with a camera on a robot arm that uses a new variant of the volume intersection approach. Due to the high performance of this algorithm, we try to use it for markerless VR immersion in a real time environment. The adaptive background model for registering the silhouettes is described in [Thu05]. The real time vertex reduction process of this project is described in [Piz06]. A rival approach [Fra04] is however based on *Bulk Synchronous Programming Scheme* by [Val90]

3. BASIC PIPELINE DESIGNS

The design of a pipeline has to consider four different aspects: the arrangement of the processed data, the communication channels between the pipeline steps, the connection topology and the controlling of the working threads.

The basic patterns presented in this chapter were inspired by [Joh04], who gives an overview of the first achievements in the nowadays disregarded dataflow processors that led to the visual dataflow programming 'languages' or tools and describes similar problems occurring during the design of a dataflow processor or a dataflow oriented software.

Another fundamental book is [Gam95], a handbook on patterns for object oriented software design. These patterns turn out to be helpful for organizing thread responsibilities as well.

The different patterns for each aspect generally describe a trade-off between flexibility, performance, parallel distribution and error proneness.

Connection Topology

Contrary to a normal dataflow graph, it is a more flexible approach to let each step decide for itself which type of data it will accept and send its results indifferently to its successors. The connection topology defines in which succeeding steps the data has to be buffered.

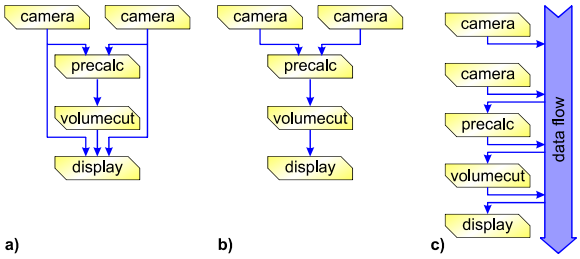


Figure 1: Different pipeline arrangements: a) typical data flow graph. b) assembly line arrangement. c) dataflow arrangement.

As shown in Figure 1, the *assembly line arrangement* is a strictly pipeline approach. As the data pass through each step of the pipeline, it has to be buffered in each of them. Unneeded data is forwarded unprocessed to its successor. In the *dataflow arrangement*, the different steps pick and buffer only their demanded data out of the data stream. The results are sent into the dataflow.

Data Arrangements

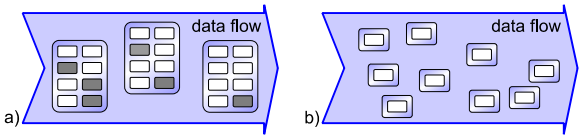


Figure 2: Different handling of the data pieces: as one complete set in the workpiece arrangement a), or free floating single data chunks b).

As shown in Fugure 2, in the *workpiece* arrangement all data is packed into one container, so that each pipeline step works on a complete data set. This arrangement might not exploit the entire potential of parallelization, but handling and overviewing is easy.

In the *free data flow* arrangement, the different types of data are sent independently through the pipeline. The pipeline steps could start working on the first incoming data part but have to take care to gather all their required data. At a first glance, this distribution might be a good idea but the effort to pick and gather all the needed data might sap all performance gain.

Communication Arrangements

First, the event that triggers a communication between the pipeline steps has to be defined. In the *data driven* concept, the steps of the pipeline become active and produce results on incoming data. In the *demand driven* concept, a step reacts on a request for its result. Whereas the first is the most used concept for data processing tasks, the latter is useful for interactive tasks.

The possibilities to layout the communication channel between the pipeline steps differ as well. The most simple way is the *newest-buffered* communication, where only the latest data is present in the channel and older data is lost. Different buffers for each type of data have to be established. This is usually the choice for visual real time applications.

Another communication arrangement is the *synchronous* communication. The channel is locked for new input until the old data is picked up by the succeeding step. This data arrangement assures that there is no data loss, but it could slow down the pipeline, or even reserialize the parallelization.

The last arrangement is the *buffering-all* construct. The sent data is parked into a buffer structure until the receiving step picks up the data for processing. Although different buffer techniques are concernable (e.g. stack, tree), in most cases a FIFO queue will be adequate. This arrangement accommodates the possibly fluctuating processing workload of the different pipeline steps. The data will not be lost and the pipeline steps do not block each other.

Thread control

active waiting	conditional waiting

DO
  IF data is waiting
  do work
  reduce sleeptime
 ELSE
  increase sleeptime
  sleep(sleeptime)
UNTIL should end
	DO
	  wait for data
	  do work
	UNTIL should end

Table 1: The working loops of an active or conditional waiting thread.

A simple pseudocode illustration of the two different thread loop designs is shown in Table 1. The *active waiting* thread repeatedly checks in its processing loop wether there is any data waiting to be processed. The thread could be put into a temporary dormant state to reduce unnecessary processing cycles. The sleeping time can increase with every unnecessary

check. The *conditional waiting* thread is initially in a waiting state and only activated if needed.

The advantage of the *conditional waiting* thread is that processor time is only used when there is effective work to do, and the thread is immediately reacting. On the other hand the *active waiting* thread is independent from the overall arrangement. All possible circumstances are local in the responsibilities of this certain thread, so the danger of deadlocks and race conditions is minimized. It is suitable for slow reacting tasks, such as user interface (e.g. rendering 24 times per second).

4. APPLIED MULTITHREADED PIPELINE DESIGN

Our reconstruction environment has typical real time conditions and requirements. There are different data types to be processed and each set of them is related to one point in time. A crucial challenge is that data parts may be missing or be obsolete due to newer data sets. Additionally, the different steps of the reconstruction process should be able to be dynamically activated during experimentation.

Data Container

Different types of data are encapsulated in a container object that holds, among other administrative data, a flag to mark if it is the last data piece with the given attributes. Usually the creating object is responsible for cleaning up its constructed data in order to avoid memory leaks during execution. To prevent complexity and bottlenecks as well as memory leaks and colliding access, we hold a strict policy: a data container and its data is accessed and in the responsibility of only one object in the pipeline at a time.

Managing the Waiting Queues

As data of different timestamps and types arrives at a pipeline step, a number of waiting queues must be handled for each communication channel between the steps. Thereby one exclusivly accessed queue collects the data of a specific timestamp. It is marked to be *full* if it received a container with a “last-one mark” for every expected data type. The queue is marked to be *done* if for every expected data type a container with the “last-one mark” was given out.

The different queues are managed by a factory construct (see Figure 3). This construct was not chosen for the typical reasons as described in [Gam95], but to create a central object to keep track of the access on its queues, similar to an inverse semaphore. So the access to a queue is monitored by a check in/ check out mechanism. This allows parallel access to different waiting queues. The competing access to one queue is managed by itself. Returned queues that are *done* and not checked out by other threads will be deleted. Queues that are older than this queue can be marked as being *done*, too. This eliminates obsolete or uncomplete queues.

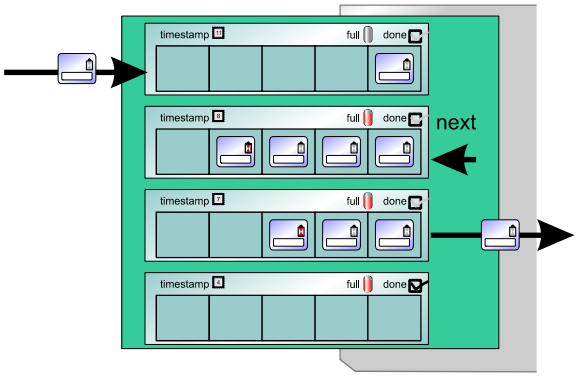


Figure 3: The queue factory for managing the different waiting queues for each timestamp.

One interesting aspect is the strategy how to determine a new queue for the receiving pipeline step after it has finished one (the next marked queue in Figure 3).

The different strategies of handling the queues resemble different aspects of the *newest-buffered* and *buffering-all* FIFO arrangements used here on complete data sets. A *synchronous* arrangement would be contradictory to the concept of a queue to uncouple sending and receiving steps. As data can be lost, and the pipeline step might wait unnecessary in a queue, it is possible to only switch to *full* queues. A compromise to reduce the waiting time may be to give priority to full queues and start processing non-*full* ones if no full one is present. Older queues are discarded and newer, but non-*full* queues are kept to be filled in the future.

In the experiments, it is well tried to keep track of the waiting full queues. Reaching a given threshold indicates that the receiving pipeline step is waiting for data that will never come, or is getting not enough processor time for its task. In the first case, the step has to discard its work on this timestamp. In the latter case, the preceeding steps have to be slowed down similar to the *synchronous* communication arrangement.

Pipeline Steps

The functional code of the pipeline step is encapsulated in different agents (similiar to the visitor pattern described in [Gam95]) to allow quick allocation of processing steps to a working thread during experimentation. As can be seen in Figure 4, each step consists of a queue management factory for the incoming data, one or more agents to process the data and a thread running through all of them. A filter might be added in front of the queue management factory to send the unprocessed data types to the succeeding steps immediately. This filter

transforms an *assembly line arrangement* into a *dataflow arrangement*.

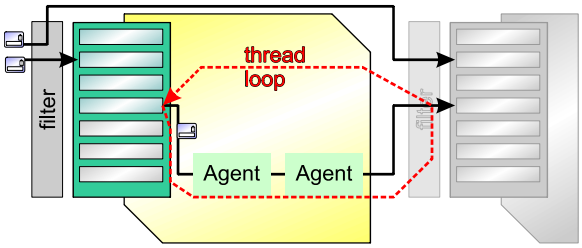


Figure 4: The pipeline step. It capsulates its incoming communication channel, its functional entities (agents) and the working thread.

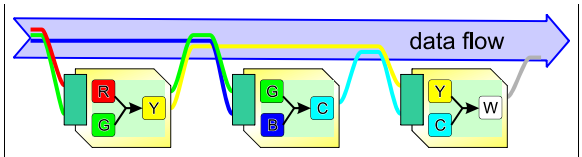
Special Pipeline Steps

The *gatherer* is a pipeline step to manage incoming data of several preceeding sending steps and set the "last-one" mark on the last container of all sending predecessors. The *distributor* has to copy the incoming data for several succeeding receiving steps.

This functionality can be implemented in a standard pipeline step but, as not every step has several predecessors or successors, there is no need for this costly administration effort.

5. THE IDLE LOAD EFFECT

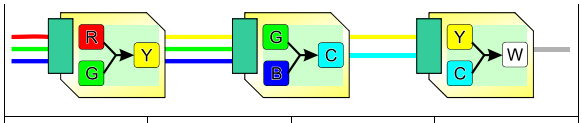
At a first glance, a *dataflow arrangement* with one of the *newest* strategies seems to distribute the work best among the pipeline steps, while minimizing administrative work.



timestamp	R+G > Y	G+R > C	Y+C > W
1	abort (no R)	-	-
2	success	abort (no B)	abort (no C)
3	abort (no R)	abort (no G)	-
4	success	abort (no B)	abort (no C)
5	abort (no R)	-	-
6	success	success	success

Table 2: Example of the idle load effect. The data flow arrangement has to abort a lot of started processing work due to missing data.

In the following example there is a *dataflow arranged* pipeline with *newest* strategy (see Table 2) in competition with a *assembly line* pipeline with *newest-full* strategy (see Table 3). There are three data sources sending data types: Red, Green, and Blue. The unreliable Red and Blue data sources shall send their data every 2nd, respectively 3rd timestamp. The three-step pipeline processes Red, Green and Blue data to the desired White data type. As can be seen easily, it can provide White data only in every 6th timestamp. But here the assembly line arrangement waits in the first step for a full queue up to the 6th timestamp, thus filtering out incomplete data sets. The supposed faster first arrangement keeps the system busy with ten futile proccessing cycles. As part of the data might be lost due to hardware or algorithmic issues or by the *newest* strategies, the data loss and futile processing work increase.



timestamp	R+G > Y	G+R > C	Y+C > W
1	discard	-	-
2	discard	-	-
3	discard	-	-
4	discard	-	-
5	discard	-	-
6	success	success	success

Table 3: Example of the idle load effect. The assembly line arrangement also has to wait for a complete data set, but does not waste processor time in the meantime.

6. REFERENCES

[Fau03] Fautz, M. Objekt- und Texturrekonstruktion mit einer robotergeführten Kamera. Shaker Verlag, 2003.

[Fra04] Franco, J.S., Ménier, C., Boyer, E. and Raffin, B. A Distributed Approach for Real Time 3D Modeling. Proceedings of the 2004 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'04) IEEE. 2004.

[Gam95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns, Addision-Wesley Publishing Company, 1995.

[Joh04] Johnston, W.M., Hanna, J.R.P. and Millar R.J. Advances in Dataflow Programming Languages. ACM Computing Surveys, Vol. 36 No. 1, ACM Press 2004.

[Lau94] Laurentini, A. The visual hull concept for silhouette-based image understanding. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 16, N. 2, 1994.

[Piz06] Pizarro, F., Preuss, S. Simplification of Reconstructed Meshes in Real Time. CASA 2006 Proceedings, Computer Graphics Society (CGS)

[Thu05] Thüring, S., Herwig, J. and Schmitt, A. Silhouette-based Motion Capture for Interactive VR-Systems including a Rear Projection Screen. CASA 2005 Proceedings, Computer Graphics Society (CGS)

[Val90] Valiant, L.G. A Bridging Model for Parallel Computation. Communication of the ACM 33(8), 1990.